Juha Ruohonen

JATKUVA INTEGRAATIO
CASE: CIMCORP OY

Tietojenkäsittelyn koulutusohjelma

2015

samk
Satakunnan ammattikorkeakoulu
Satakunta University of Applied Sciences

JATKUVA INTEGRAATIO CASE: CIMCORP OY

Ruohonen, Juha
Satakunnan ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Maaliskuu 2015
Ohjaaja: Nieminen, Hans
Sivumäärä: 36
Liitteitä: 0

Asiasanat: jatkuva integraatio, testaus, versionhallinta, Jenkins CI

Tämän opinnäytetyön tarkoituksena oli toteuttaa integraatiotestejä automaattisesti suorittava jatkuvan integraation järjestelmä Cimcorp Oy:lle. Opinnäytetyö tehtiin yrityksen tuotteen kehitystä ja testausta varten yrityksen toiveesta.

Järjestelmän pohjana oli jatkuvan integraation työkalu Jenkins CI. Kun järjestelmä saa tiedon muutoksesta versionhallintajärjestelmältä, kääntää se automaattisesti lähdekoodin ja ajaa integraatiotestit. Työkalu myös muodostaa raportin testituloksista ja tarvittaessa lähettää muutoksen tehneelle ohjelmistokehittäjälle ilmoituksen sähköpostitse. Tärkeää on myös pystyä tekemään kuvattu prosessi useammalle projektille kerrallaan, mikä edellytti myös jotain lähdekoodimuutoksia tuotteeseen.

Tuloksena syntyi edellä kuvatut toiminnot toteuttava jatkuvan integraation järjestelmä. Valmis järjestelmä ei toteuttanut kaikkia toivottuja lisäominaisuuksia, mutta hyväksyttiin silti vaatimukset toteuttavana kokonaisuutena.

CONTINUOUS INTEGRATION CASE: CIMCORP OY

Ruohonen, Juha
Satakunnan ammattikorkeakoulu, Satakunta University of Applied Sciences
Degree Programme in Business Information Systems
March 2015
Supervisor: Nieminen, Hans
Number of pages: 36
Appendices: 0

Keywords: continuous integration, testing, version control, Jenkins CI

The purpose of this thesis was to create a continuous integration system that would automatically run integration tests. This thesis was made for the development and testing of Cimcorp Oy's product at the company's request.

The basis of the system was Jenkins CI, a tool for continuous integration. The system automatically compiles the source code and runs the integration tests when it detects a change in the version control system. The system also constructs a test report and if necessary, sends an email notification to the developer who made the changes. It is also important that this process can be done to multiple projects simultaneously, which required some changes in the product's source code.

A system of continuous integration which implements the aforementioned was the result. The system did not fulfill all additional requests, but was nevertheless accepted as a result that met the requirements.

SISÄLLYS

TERMIT JA LYHENTEET

Build	Tietokoneohjelman versio tai sen ”rakentaminen”, lähdekoodin kääntäminen ja kokonaisuuden kokoaminen jakelukelpoiseen muotoon
CI	Continuous Integration, Jatkuva Integraatio
CIServ(er)	Työhön liittyvä Jenkinsin asennus
Haara	Ohjelmakoodin versio (Git-)repositoriossa
Jenkins	Jenkins CI, jatkuvaan integraatioon suunniteltu työkalu
LTS	Long-term support, tavallista pidempään tuettu versio ohjelmistosta. Jenkinsin yhteydessä LTS on harvemmin päivitettävä vaihtoehto
Refaktorointi	Ohjelmakoodin järjestäminen uudelleen toiminnallisuutta muuttamatta
Repositorio	Versionhallintajärjestelmien käyttämä varasto
SSH	Secure Shell, salatun tietoliikenteen protokolla
XML	Extensible Markup Language, merkintäkieli
XP	Extreme Programming, ketterä ohjelmistokehitysmenetelmä

1 JOHDANTO

Ohjelmistokehitys on vaativaa ja aikaa vievää työtä. Perinteisiä ohjelmistokehitysmenetelmiä kuten vesiputousmallia käyttäen jää kehitystyön loppuun lisäksi pitkä integraatiojakso, mistä aiheutuu odottamattomia kuluja sekä viivästyksiä. Muutosten tekeminen kehitystyön lopussa on huomattavasti kalliimpaa kuin sen alussa. Näistä syistä johtuen Cimcorp Oy:n tuotteen CellController kehitystyössä käytetään ketteriä menetelmiä, joista jatkuva integraatio eli continuous integration varmistaa järjestelmän yhteistoiminnallisuuden joka päivä. Jatkuvalla integraatiolla myös voidaan taata parempi toimintavarmuus, mikä on automaatiojärjestelmissä hyvin tärkeää.

Opinnäytetyön perimmäinen tarkoitus on tuottaa Cimcorp Oy:lle jatkuvan integraation ympäristö, missä tuotteen CellController integraatiotestit ajetaan automaattisesti järjestelmän havaittua muutoksen tuotteen lähdekoodissa. Alkutilanteessa integraatiotestit suoritettiin tuotekehittäjien toimista omilla työkoneilla. Koska testien suorittaminen on hidasta, kuluu siihen työaikaa joka olisi muuten voitu käyttää muihin tehtäviin. Testien suorittaminen oli myös epäsäännöllistä, mikä ei ole hyvä tuotekehityksen kannalta.

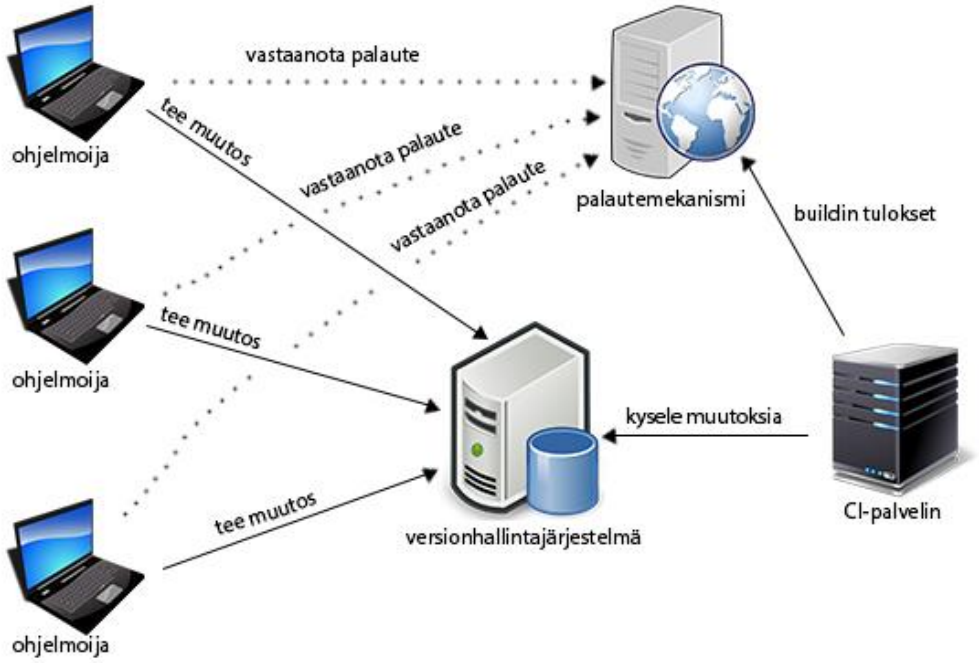
Opinnäytetyö tehtiin Cimcorp Oy:n pyynnöstä. Cimcorp Oy on vuonna 1975 perustettu automaatiojärjestelmiä rakentava ja toimittava yritys, jonka nykyisin omistaa japanilainen yritys Murata Machinery Ltd. Cimcorp oy työllistää yli 200 henkilöä ja sen markkina-alueita ovat pääasiassa Eurooppa, Aasia ja Pohjois-Amerikka.

2 JATKUVA INTEGRAATIO

Ohjelmistotuotannossa integraatiolla tarkoitetaan ohjelmiston osien kokoamista toimivaksi kokonaisuudeksi. Tässä luvussa kerron mitä on jatkuva integraatio ja sen toimintaperiaatteista. Käyn lyhyesti läpi myös jatkuvan integraation historiaa.

2.1 Mitä on jatkuva integraatio?

Continuous Integration (CI) eli jatkuva integraatio on ohjelmistokehitysmenetelmä missä kehitystiimin jäsenet integroivat tekemänsä koodimuutokset usein, yleensä vähintään kerran päivässä. Tällä toimintaperiaatteella integrointi tapahtuu useita kertoja päivässä. Jokainen integrointi todennetaan eli verifioidaan automaattisella buildilla mikä sisältää myös testauksen, jotta virheet löytyisivät mahdollisimman nopeasti (Kuva 1). Tällä tavoin voidaan vähentää integraatioon liittyviä ongelmia, jolloin kehitystyötä voidaan jatkaa nopeammin. (Fowler 2014; Kawarelowicz & Berntson 2011, 5-6.)



Kuva 1. Jatkuva integraatio -järjestelmän toiminta. ClipArt-kuvat ovat vapaasti käytettäviä ilmaiskuvia.

Jatkuvaa integraatiota voidaan pitää nykyaikaisen ohjelmistokehityksen kulmakivenä. Kun jatkuva integraatio otetaan käyttöön organisaatiossa, muuttaa se tiimien ajattelutavat koko kehitysprosessista. Hyvä jatkuvan integraation infrastruktuuri voi tehostaa kehitystä aina tuotantoon asti, auttaen löytämään ja korjaamaan virheet nopeammin sekä tarjoten hyödyllisen projektikojelaudan kehittäjille ja muille osallisille. (Smart 2011, 1.)

Yksinkertaisimmassa muodossaan jatkuva integraatio sisältää työkalun, joka tarkkailee versionhallintajärjestelmää muutosten varalta. Muutoksen havaitessaan työkalu automaattisesti kääntää koodin ja testaa ohjelman. Yhden tai useamman virheen havaitessaan työkalu välittömästi ilmoittaa kehittäjälle, jotta ongelma voidaan korjata mahdollisimman nopeasti. Jatkuva integraatio voi tehdä kuitenkin paljon enemmän. Se voi auttaa pitämään koodipohjan terveyden tarkkailussa ja automaattisesti monitoroida koodin laatua ja kattavuutta, sekä auttaa pitämään tekniset vaatimukset ja ylläpitokustannukset alhaisina. Työkalu voi myös toimia kehitystyön tilan mittarina. Se voi myös yksinkertaistaa ja nopeuttaa toimitusta, sallien kehittäjien ottaa käyttöön kehitetyn ohjelmiston viimeisin versio joko automaattisesti tai yhden napin painalluksella. Perusolemukseltaan jatkuva integraatio on riskien vähentämistä tarjoamalla nopeampaa palautetta. Ensisijaisesti se on suunniteltu auttamaan integraatio- ja taantumusongelmien nopeammassa huomaamisessa sekä korjaamisessa, mistä seuraa sujuvampi ja nopeampi toimitus vähemmin virhein. Lisäksi käyttöönoton automatisoimalla on ohjelmisto helpompi toimittaa testaajille sekä loppukäyttäjille nopeammin, luotettavammin ja vähemmällä vaivalla. (Smart 2011, 1-2.)

2.2 Historiaa

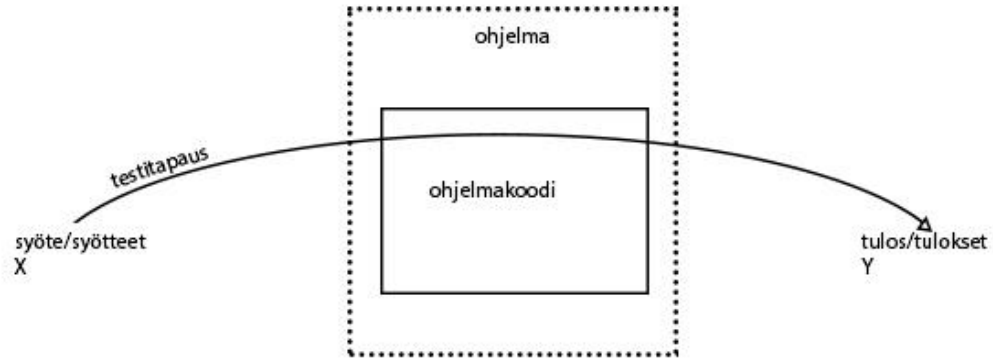
Ennen jatkuvana integraationa tunnettua ohjelmistokehitysmenetelmää kului kehittäjätiimeillä paljon aikaa ja voimavaroja julkaisua edeltävässä integraatiovaiheessa. Tässä vaiheessa yksittäisten kehittäjien tai pienten tiimien koodimuutokset tuotiin yhteen ja kasattiin toimivaksi tuotteeksi. Työ oli raskasta, joskus sisältäen kuukausia ristiriitaisten muutosten integrointia. Oli vaikeaa ennakoida kasautuvia ongelmia ja vielä vaikeampaa korjata niitä, sillä tämä yleensä tarkoitti viikkoja tai kuukausia sitten kirjoitetun koodin uudelleentyöstämistä. Kivulias prosessi johti usein merkittäviin toimitusviivästyksiin, suunnittelemattomiin kustannuksiin ja seurauksena tyytymättömiin asiakkaisiin. Jatkuva integraatio syntyi näiden ongelmien vähentämiseksi. (Smart 2011, 1.)

3 TESTAUS

Tässä luvussa kuvailen testausta ja sen tasoja. Käsittelen myös lyhyesti ohjelmistovirheitä ja niihin liittyen surullisenkuuluisan esimerkkitapauksen.

3.1 Mitä on testaus?

Puhekielessä termillä testaus tarkoitetaan yleensä kokeilua liittyen melkein mihin tahansa. Perinteinen ohjelmistotestaus voidaan määritellä virheiden suunnitelmalliseksi etsimiseksi suorittamalla ohjelmaa tai jotakin sen osaa. Testausta käytetään myös keinona todistaa ohjelmiston toimivuus. Testaus on käytännössä testitapausten suorittamista testattavalla ohjelmistolla. Testitapaukset ovat osa ohjelmakoodia, tehty täysin testaamista varten. Testitapauksella on syötteitä sekä yksi tai useampi oletettu lopputulos, minkä perusteella testitapaus on joko onnistunut tai epäonnistunut (Kuva 2). Tulokset saattavat ja niiden usein pitäisikin vaihdella syötteiden sekä ohjelman sisäisen tilan perusteella. Ohjelman sisäiseen tilaan vaikuttavat muun muassa muuttujien arvot sekä mahdollinen ohjelman ulkopuolelta – esimerkiksi kiintolevyltä tai tietokannasta – haettu ja käsitelty data. Ohjelmistotestauksen pääpiirre on määrittää joukko testitapauksia testattavalle ohjelmistolle. (Haikala & Märijärvi 2006, 284-285; Jorgensen 2008, 4-5.)



Kuva 2. Testaus yksinkertaistettuna (Haikala & Märijärvi 2006, 284).

3.2 Mitä ovat virheet?

Ihmiset tekevät virheitä. Kun virhe tapahtuu ohjelmoinnissa, käytetään siitä nimitystä bugi. Virheillä on tapana kasaantua. Vaatimuksiin liittyvä virhe saattaa muuttua pahemmaksi suunnittelussa ja edelleen vahvistua ohjelmoitaessa. Virheistä syntyy vikoja. On tarkempaa sanoa vian olevan virheen esiintymä. Tällaisia vikoja voidaan myös kutsua bugeiksi. Häiriö tapahtuu kun vika suoritetaan ohjelmakoodia ajettaessa, mistä voi seurata esimerkiksi ohjelman kaatuminen. Termi bugi otettiin tietotekniikassa käyttöön vuonna 1947 Harvardin yliopistossa, kun silloinen huippuluokan tietokone Mark II yllättäen sammui. Syyksi paljastui tietokoneeseen lentänyt hyönteinen. (Jorgensen 2008, 3-4; Patton 2006, 9.)

Bugit voivat olla niinkin huomaamattomia, ettei voi olla varma onko kyseessä bugi vai ei. Toisessa ääripäässä bugi voi johtaa henkivahinkoihin. Bugi saattaa olla hyvinkin halpa korjata, mutta korjauksen levitys voi maksaa miljoonia. Useimmat bugit ovat vaikeasti löydettäviä ja sinällään harmittomia, eikä osaa niistä korjata koskaan. Esimerkkinä huomattavammasta bugista voidaan käyttää NASAn Mars Polar Lander -avaruusluotainta, joka katosi joulukuun kolmantena päivänä vuonna 1999 laskeutuessaan Marsin pinnalle. Tutkinnan kautta pääteltiin epäonnistumisen todennäköisimmäksi syyksi yhden bitin odottamaton asetus. Hälyttävintä oli, ettei ongelmaa havaittu sisäisissä testeissä. Teoriassa luotaimen olisi pudotessaan pitänyt avata laskuvarjo hidastaakseen pudotusta ja muutamaa sekuntia laskuvarjon avaamisen jälkeen luotaimen jalkojen piti siirtyä laskeutumisasentoon. 1800 metrin korkeudella laskuvarjo irrotetaan ja luotain laskeutuisi pinnalle laskeutumismoottoreiden avulla. (Patton 2006, 11, 13.)

Säästääkseen rahaa NASA yksinkertaisti moottoreiden sammutusmekanismia käyttämällä kontaktikatkaisijaa luotaimen jalassa, mikä asettaisi tietokoneessa bitin tilaan mikä käskee sammuttamaan moottorit. Moottorit siis yksinkertaisesti sammuisivat luotaimen laskeuduttua planeetan pinnalle. Tutkijat havaitsivat testeissään, että luotaimen jalkojen avautuessa mekaaninen tärinä laukaisi myös katkaisimen asettaen bitin sammutusarvoon. On hyvin todennäköistä että luotain luuli jo laskeutuneensa ja sammutti moottorit, minkä jälkeen luotain levisi kappaleiksi osuttuaan pintaan 1800 metrin vapaan pudotuksen jälkeen. Tulos oli katastrofaalinen,

mutta syy yksinkertainen. Luotainta testasi useampi tiimi. Yksi tiimi testasi jalkojen avautumisen ja toinen tiimi testasi laskeutumisprosessin loppuosan. Ensimmäinen tiimi ei koskaan tarkistanut oliko laskeutumisbitti asetettu, sillä se ei kuulunut tiimin vastuualueeseen. Toinen tiimi taas aina nollasi tietokoneen ennen omia testejään, milloin myös kyseinen bitti nollautui. Molemmat osat toimivat täydellisesti erikseen, mutta eivät enää yhdessä. (Patton 2006, 11-12.)

3.3 Testaustasot

Ohjelmistotestaus voidaan jakaa kolmeen selkeään tasoon: yksikkö-, integraatio- ja järjestelmätestaus. Jokaisella tasolla on omat ongelmat ja päämäärät. Yksikkötestaus keskittyy yksittäiseen moduuliin tai luokkaan, mikä koostuu tyypillisesti noin 100-1000 koodirivistä. Yksikkötestausta varten voidaan joutua simuloimaan ohjelman ympäristöä tai muita moduuleita, esimerkiksi kohdeympäristön antureita tai laitteita. (Haikala & Märijärvi 2006, 288-289; Jorgensen 2008, 201.)

Integraatiotestausta varten yhdistetään moduuleita tai moduuliryhmiä. Päämääränä on tutkia moduulien välisten rajapintojen toimivuutta. Yksikkötestaus saatetaan tehdä rinnakkain integraatiotestauksen yhteydessä, mutta testauksen kattavuuden ja selkeyden kannalta on suotavaa suorittaa yksikkötestaus ensin loppuun. Tyypillisesti integrointi tapahtuu alimman tason moduuleista ylöspäin eli kokoavasti. Jäsentävässä tai osittavassa integraatiossa aloitetaan ylimmän tason moduuleista, eli järjestys on päinvastainen. (Haikala & Märijärvi 2006, 290; Jorgensen 2008, 311.)

Järjestelmätestaus on koko järjestelmän testausta, missä tuloksia verrataan usein asiakasdokumentaatioon ja odotuksiin. Tarkoituksena ei tyypillisesti olekaan löytää virheitä vaan todistaa oikeanlainen toiminta. Järjestelmätestaukseen saattaa liittyä myös kenttä- ja hyväksymistestaus. Kolmesta mainitusta tasosta järjestelmätestaus on lähimpänä jokapäiväistä elämää, kuten esimerkiksi auton koeajo. Järjestelmätestauksessa testataan myös ei-toiminnallisia ominaisuuksia kuten luotettavuus, käytettävyys ja kuormitus. Virheiden löytäminen järjestelmätestauksen aikana tulee helposti kalliiksi, sillä muutoksia voi joutua tekemään useaan moduuliin minkä seurauksena yksikkö- ja integraatiotestit täytyy myös suorittaa uudelleen ennen

uutta järjestelmätestausta. Tällaisesta uudelleentestauksesta käytetään termiä regressiotestaus, riippumatta testaustasosta. (Haikala & Märijärvi 2006, 290; Jorgensen 2008, 229.)

3.4 Lähestymistapoja

Kaksi termiä, millä testauksen lähestymistapoja kuvataan, ovat mustalaatikkotestaus (black box testing) eli funktionaalinen testaus ja lasilaatikkotestaus (glass/clear/white box testing) eli rakenteellinen testaus. Mustalaatikkotestauksessa ohjelman toimintaa ei tunneta, vaan testitapaukset valitaan spesifikaation perusteella. Lasilaatikkotestauksessa hyödynnetään tietoa ohjelman toteutuksesta ja testitapaukset suunnitellaan sen perusteella. (Haikala & Märijärvi 2006, 291; Patton 2006, 55.)

Kaksi muuta käytettyä termiä ovat staattinen ja dynaaminen testaus. Dynaaminen testaus on mitä normaalisti testauksesta ajatellaan, ohjelman suorittamista ja käyttämistä. Staattinen testaus liittyy vahvasti mustalaatikkotestaukseen, eli spesifikaatiota vasten testaamiseen. Spesifikaatio on dokumentti, ei ajettava ohjelma, joten sitä pidetään staattisena. Termit ovat helppoja ymmärtää vertauksen kautta. Käytettyä autoa tutkittaessa renkaiden kunnon ja maalipinnan tutkiminen sekä moottorin tarkastelu ovat staattisia testaustapoja. Auton käynnistäminen, moottorin kuuntelu ja koeajo ovat dynaamisia testaustapoja. (Patton 2006, 56.)

4 KETTERÄ OHJELMISTOKEHITYS

Tässä luvussa kerron ketterästä ohjelmistokehityksestä ja sen menetelmistä. Tämän työn huomiossa ovat pääasiassa kaksi ketterää menetelmää: Extreme Programming (XP) ja Scrum.

4.1 Yleistä

Minkä tahansa ketterän ohjelmistokehitysprosessin pitää pystyä mukautumaan nopeasti tapahtuviin muutoksiin projektin laajuudessa ja vaatimuksissa, mutta sen täytyy myös pystyä toimittamaan korkealaatuinen lopputulos tavalla joka on kustannustehokas eikä kärsi turhaan byrokratiasta. Prosessi ei myöskään saa vaatia osallistuvilta kehittäjiltä mahdottomuuksia. Ketterän ohjelmistokehitysmenetelmän pitää täyttää joitain perustavaa laatua olevia ominaisuuksia. Ohjelmistokehityksen pitää mukautua asiakkaan ongelman muuttuessa. Toimitetun tuotteen tulisi myös sallia mahdolliset myöhemmät kehitystarpeet. Ohjelmiston laadunvarmistus on tärkeää. Miten voidaan varmistaa ohjelmiston aina toimivan, kuten sen pitäisi toimia? Yhtenä huolenaiheena on myös tarpeeton dokumentointi ja muu byrokratia, millä kehitysprosessia yleensä ylläpidetään ja hallinnoidaan. (Holcombe 2008, 2.)

Ohjelmistokehitysprojektia aloitettaessa ensimmäinen ja joidenkin mielestä vaikein vaihe on selvittää yhdessä asiakkaan kanssa, mitä valmiin järjestelmän tulisi tehdä. Kun jonkinlainen kokonaiskäsitys järjestelmän tarkoituksesta on tiedossa, voidaan seuraavaksi miettiä miten järjestelmä on vuorovaikutuksessa muiden liiketoimintaprosessien kanssa. Tässä vaiheessa myös pyritään rajaamaan järjestelmä. Näiden tietojen perusteella luodaan yksityiskohtainen määrittelydokumentti. Asiakas saattaa tässä vaiheessa olla tyytyväinen esitettyyn ratkaisuun, mutta kenelläkään ei ole vielä tarkkaa käsitystä miten järjestelmä toimisi tässä vaiheessa ja sitä ei välttämättä ole ymmärretty täysin oikein. Tämän jälkeen alkaa analysointi mistä siirrytään suunnitteluun. Vaihe on yleensä pitkä ja monimutkainen. Esiin tulevia ongelmia ei välttämättä keskustella asiakkaan kanssa vaan kehittäjät tekevät päätökset itse. Järjestelmä saattaa alkaa ajautumaan pois siitä, mitä sen pitäisi olla. Prosessin lopussa yksityiskohtainen suunnitelma on suuri ja monimutkainen. Suunnitelma on mahdollisesti vielä pätevä suhteessa asiakkaan liiketoimintatarpeisiin, mitkä ovat voineet kehittyä. Jos tässä vaiheessa palataan asiakkaan luo, on mahdollista että yritys tai sen toiminta on muuttunut jollain tapaa ja vaatimukset ovat jo huomattavasti erilaiset. Perinteiset ohjelmistokehitysmenetelmät kuten esimerkiksi vesiputousmalli eivät voi vastata haasteeseen tehokkaasti. Koska suunnitteluun on jo investoitu paljon, voi olla epämiellyttävää lähteä muuttamaan suunnitelmaa tai aloittaa alusta.

Tämäntyyppiset mallit eivät ole kovin hyviä reagoimaan muuttuviin tarpeisiin. (Holcombe 2008, 2-4.)

Ketterissä menetelmissä käytetyt iteraatiot määrittelystä ohjelmointiin ja testaukseen asti ovat tyypillisesti lyhyitä. Kehittämisen voi jopa ajatella olevan jatkuvaa. Tämän tekee mahdolliseksi testitapausten suorittamisen automatisointi. Yksinkertaistettuna prosessi ohjelmamuutoksen tekemiseen tai uuden ominaisuuden lisäämiseen etenee seuraavasti. Tehtävälle muutokselle ohjelmoidaan yksi tai useampi testitapaus, mitkä raportoivat testien epäonnistumisen ajettaessa. Muutosten ohjelmointi alkaa vasta tässä vaiheessa ja jatkuu niin kauan että kaikki testitapaukset, mukaan lukien aiemminkin tehdyt, menevät virheettömästi läpi. (Haikala & Märijärvi 2006, 47.)

4.2 Extreme Programming

Kent Beckin luoma Extreme Programming eli XP-menetelmä on ehkäpä tunnetuin ketterä menetelmä. Menetelmällä on viisi keskeistä arvoa, mitkä ovat syy sen olemassaoloon sekä menestymiseen: kommunikaatio, palaute, yksinkertaisuus, rohkeus ja kunnioitus. Menetelmä pyrkii toimintatavoillaan painottamaan tehokasta kommunikaatiota kaikkien asianomaisten välillä. Asiakas pidetään ajan tasalla ja mahdollisimman hyvin mukana projektissa. Tällöin pysyy jonkinlainen varmuus työn edistymisestä sekä siitä, että asiakkaalle ollaan tekemässä tarpeita vastaavaa järjestelmää. Järjestelmä pyritään pitämään mahdollisimman yksinkertaisena, harkiten tarkasti mitä ominaisuuksia todella tarvitaan. Joskus monimutkaiset ratkaisut ovat tarpeellisia, mutta nekin on oltava perusteltavissa ja testattavissa. Tässä yhteydessä rohkeudella tarkoitetaan itsevarmuutta tehdä asioita, joita voitaisiin muuten pitää riskeinä. Tarpeet voivat muuttua projektin aikana, jolloin aiemmin tehty työ voi vaatia muutoksia. On luonnollista vastustaa tällaisia muutoksia perinteisiä ohjelmistokehitysmenetelmiä käyttäen. Uusista haasteista nauttiminen on osa XP:n filosofiaa. Kuten vuorikiipeily ilman köyttä, ohjelmistokehitys ilman suunnitelmaa vaikuttaa aluksi itsetuhoiselta. Menetelmällä on kuitenkin rajoituksia ja sen tapoja tulee seurata. Monet ohjelmistokehityksessä esiintyvät ongelmat ovat niin sanottuja ihmisongelmia eivätkä teknisiä ongelmia. Jokaista yksilöä kohdellaan kunnioittavasti ja mielipiteet sekä eri näkökulmat otetaan huomioon, vastuuta jaetaan ja luotetaan

kaikkien tekevänsä osuutensa järkevästi. XP on ihmismäinen tapa tehdä asioita. (Haikala & Märijärvi 2006, 47; Holcombe 2008, 19-25.)

XP-menetelmässä on kaksitoista olennaista käytäntöä. Ensimmäinen niistä on testien tekeminen ennen varsinaisen ohjelmoinnin aloittamista. Tavoitteena on jatkuva testaus. Testit epäonnistuvat koska koodia ei ole kirjoitettu. Testauksessa on tärkeää, että kaikki testitapaukset ajetaan joka kerta. Testitapaukset ovat tärkein resurssi ja niitä parannetaan jatkuvasti. Testit tavallaan korvaavat spesifikaation ja suunnittelun. Ne tarjoavat nopean palautemekanismin mikä kertoo, onko koodi tehty niin sanotusti oikein. Jos mikä tahansa testi epäonnistuu, on koodi korjattava. Tässä käytännössä on kuitenkin myös ongelma, sillä testaukselle pyhitettyjä kursseja ei kouluissa juurikaan ole ja ohjelmointia opetettaessa testaus jää usein huomiotta. Tästä johtuen ohjelmoijan tekemät testit eivät välttämättä ole tarpeeksi kattavia. Testit ovat kuitenkin ehdottoman tärkeitä, sillä kehittäminen pohjautuu testeihin tässä menetelmässä. Kuka tahansa voi tehdä testitapauksia, mutta vain älykkäimmät kehittäjät tekevät todella hyviä testejä. Tämän lisäksi testitapaukset kirjoitetaan ennen ohjelmointia aiheuttaen lisää ongelmia monille testaustekniikoille, kuten esimerkiksi lasilaatikkotestaus missä ohjelmarakenne tunnetaan. (Holcombe 2008, 25-26.)

Pariohjelmointi on toinen avainominaisuus. Projekti organisoidaan siten, että työ tehdään pareissa. Yksi henkilö käyttää näppäimistö toisen tarkkaillessa näyttöä ja molemmat keskustelevat mitä tekevät. Kaikki ohjelmakoodi kirjoitetaan tällä tavalla. Tämä prosessi on jatkuvaa katselmointia ja varmistaa, että virheitä tehdään harvemmin. Syy asioiden tekemiseen tietyllä tavalla on myös avoin keskustelulle. Tapa ei päde ainoastaan ohjelmointiin, vaan sitä tulisi käyttää XP-projektin jokaisessa osa-alueessa. Ihmiset tekevät työtä yhdessä, kooten osaamisensa ja älynsä sekä jakaen tietoa. Projektin suunnittelu ja keskustelu asiakkaan kanssa tulisi myös sisältää niin monta tiimin jäsentä kuin mahdollista. Parit vaihtuvat säännöllisesti. Roolien vaihto pareissa ja kehittäjien vaihtaminen parista toiseen antaa paremman käsityksen, mitä projektissa tehdään. Se on myös hyvä tapa oppia: parina oleva henkilö saattaa olla asiantuntija projektin jossain osa-alueessa tai käytetyssä tekniikassa. Yksi ero perinteisen ohjelmistokehityksen ja XP:n välillä onkin se, että XP-tiimit ovat aina puhumassa toisilleen. Perinteisiä menetelmiä käyttävät toimistot ovat verrattain

hiljaisia kun kaikki keskittyvät hiljaa omiin näyttöihinsä ja mahdollinen, vähäinen puhe ei välttämättä liity projektiin. (Holcombe 2008, 26-27.)

Asiakkaan läsnäolo on suositeltavaa projektin edetessä. Tämä rikastaa keskustelua kehittäjätiimin ja asiakkaan välillä. Asiakas voi myös paremmin ohjata projektin suuntaa ja määrätä tärkeysjärjestyksen. Kuten pariohjelmoinnissakin, tämä XP:n osa-alue suosii henkilökohtaista kanssakäymistä ja keskustelua. (Holcombe 2008, 27-28.)

Asiakas kuvailee liiketoimintaprosesseja, minkä perusteella pyritään tunnistamaan pieniä paloja ohjelmoitavia toiminnallisuuksia. Tämän perusteella ohjelmoidaan lyhyessä ajassa toimiva osa ohjelmaa. Viikko on ajallisesti hyvä tavoite. Asiakkaan pitää pystyä ymmärtämään toiminnallisuus sekä sen osa järjestelmässä. Projektin tilanne tulisi katselmoida asiakkaan ja käyttäjien kanssa säännöllisesti noin kerran kuukaudessa. Toiminnallisuuksia ei kuitenkaan heti toteuteta. Kun tarkoituksenmukaisia toiminnallisuuksia on päätelty tarpeeksi ja niiden toteuttamiseen kuluva aika ja resurssit ovat arvioitu, valitsee asiakas toteutettavaksi liiketoiminnan kannalta suurinta arvoa tarjoavat toiminnallisuudet. (Holcombe 2008, 28-29.)

Rakennettavat toiminnallisuudet on valittu ja järjestelmän luonti voidaan aloittaa. Ensimmäiseksi tapahtuu testitapausten tekeminen, mihin saadaan vihjeitä toiminnallisuuksista. Järjestelmä on tässä vaiheessa metafora joka rakennetaan organisoimalla kokoelma luokkia ja metodeita, millä toiminnallisuudet pystytään toteuttamaan. Toiminnallisuudet tehdään itsenäisiksi ja toimitettaviksi osiksi ohjelmistoa, mitä päätösten tulisi tukea. Aluksi järjestelmä voi olla epämääräinen, mutta osat kiinteytyvät nopeasti ja ovat sen jälkeen tarkemmin dokumentoitavissa. (Holcombe 2008, 29-30.)

Lyhyet iteraatiot ovat tärkeä osa XP-menetelmää. Julkaisuja tapahtuu usein. Kun on tuotettu jotain liiketoiminnalle hyödyllistä, se toimitetaan ja asennetaan asiakkaalle. Tämä antaa käyttäjille mahdollisuuden tutkia sitä ja antaa asiakkaan kautta palautetta kehittäjille. Monessa tapauksessa löytyy yksinkertaisia käyttöliittymäparannuksia jotka voidaan tehdä tai jotka antavat kuvaa siitä, miten järjestelmäkokonaisuus voi tukea liiketoimintaa. Tämä voi aiheuttaa muutoksia projektin laajuudessa tai vaatimuksissa ja on siten tärkeää kehittäjille. Julkaisuja ei kuitenkaan pidetä

prototyyppeinä, vaan ne ovat kaikki oikeita julkaisuja. Jokainen julkaisu on toiminnallisesti hyödyllinen ja toteuttaa enemmän toiminnallisuuksia. Jokainen julkaisu on myös läpikotaisin testattu. (Holcombe 2008, 30.)

Kuten jo aiemmin mainittu, ratkaisujen tulisi olla yksinkertaisia. Jos ominaisuus ei ole asiakkaalle välttämätön, sitä ei tulisi toteuttaa. Ohjelmoijalle tämä voidaan käsittää minimimääränä luokkia ja metodeita, joilla testit menevät läpi. Ohjelmakoodin yksinkertaisuus ei kuitenkaan aina tarkoita toiminnallisuuden yksinkertaisuutta. (Holcombe 2008, 30-31.)

Jatkuva integraatio on tärkeä osa XP-menetelmää ja on tärkeä itsevarmuuden lähde tiimille siitä, että projekti etenee (Holcombe 2008, 31). Jatkuvaa integraatiota käsitellään tarkemmin luvussa 2.

Ohjelmointistandardit määrittävät tiimin eri jäsenten tekemän koodin kommunikoinnin keskenään. Kaikkien tulisi käyttää samaa ohjelmointityyliä ja vastaavasti nimetä luokat ja metodit kaikkien tuntemien protokollien mukaan. Koska lähdekoodi kuvaa paljon, on erityisen tärkeää että kaikki ymmärtävät sitä hyvin. On suositeltavaa käyttää XML:ää ymmärtämisen ja rakenteellisuuden auttamiseksi. (Holcombe 2008, 32.)

Kaikki ohjelmakoodi kuuluu kaikille ohjelmoijille. Kuka tahansa voi muuttaa mitä tahansa. Tämä on kiistanalainen osa XP-menetelmää joka tuntuu menevän vasten tervettä järkeä ja perinteisiä käytäntöjä. Koodin samankaltaisuuden ansiosta tiimin jokaisen jäsenen pitäisi ymmärtää mikä tahansa osa koodia, sen tarkoituksen ja miten se liittyy kokonaisuuteen. Refaktorointia käytetään pääasiassa koodin yksinkertaistamiseen, jotta se on paremmin ymmärrettävissä ja näin ollen ylläpidettävissä. (Holcombe 2008, 32-33.)

Väsyneet ohjelmoijat kirjoittavat huonoa koodia ja tekevät enemmän virheitä. Ohjelmistoteollisuudessa ollaan usein riippuvaisia yksilöiden sankarillisista teoista. Ohjelmoija joutuu tekemään töitä enemmän kuin kohtuullisen työajan puitteissa on mahdollista. Ei ole siis ollenkaan yllättävää, että virheitä syntyy. Tästä juoksumattomaisesta lähestymistavasta on päästävä pois. XP:n on organisointitapa

auttaa vähentämään muun muassa epärealistisista aikatauluista, integroinnista sekä loppuun jätetystä testauksesta aiheutuvaa stressiä. XP:n on tarkoitus onkin minimoida tämänlainen stressi minimoimalla ylitöiden tarpeen. XP:n kannattajat väittävät sen tarjoavan pitkäjänteisemmän tavan kehittämiseen, sallien tasaisen tahdin ja paremman laadun sekä suuremman työtyytyväisyyden. Koska edistystä tapahtuu huomattavasti enemmän, voidaan työaikojakin lyhentää. (Holcombe 2008, 33-34.)

4.3 Scrum

Scrum on hallintaprosessi jota voidaan soveltaa moniin eri aktiviteetteihin, ei ainoastaan ohjelmistokehitykseen. Projekti jaetaan ominaisuuksiin ja jokaiselle ominaisuudelle määrätään arvo sekä arvioitu työpanos tai kustannukset. Projekti etenee iteraatioihin verrattavissa olevilla aikarajatuilla sprinteillä. Päivittäin pidettävät Scrum-palaverit ovat lyhyitä ja ytimekkäitä sekä auttavat hahmottamaan projektin etenemistä. Jokaisen sprintin lopussa pidetään sprintin katselmointi jossa arvioidaan tuotettujen ominaisuuksien laatua. Scrumia voidaan käyttää yhdessä jonkin muunkin ketterän menetelmän kanssa. Scrumia saatettaisiin käyttää esimerkiksi projektin yleiseen hallinnoimiseen, kun jotain muuta ketterää menetelmää käytettäisiin tuotantoon. (Holcombe 2008, 15.)

Scrum-tiimiä johtaa ScrumMaster-roolissa oleva henkilö, yleensä projektiesimies tai projektipäällikkö. ScrumMaster on vastuussa siitä että Scrumin arvoja, käytäntöjä ja sääntöjä noudatetaan. Noin viisitoista minuuttia kestävissä päivittäisissä palavereissa ScrumMaster kuuntelee tiiminsä raportit ja vertaa saavutettuja tuloksia odotettuihin tuloksiin, mitkä perustuvat sprintin tavoitteisiin sekä edellisissä palavereissa tehtyihin arvioihin. Jos joku on esimerkiksi työskennellyt saman yksinkertaisen tehtävän kanssa useamman päivän, tarvitsee hän luultavasti apua. ScrumMaster pyrkii hahmottamaan tiimin etenemisnopeuden: onko tiimi tai jokin sen jäsen jumissa vai eteneekö työ hyvin? Tiimin tarvitessa apua ScrumMaster tapaa tiimin ja pyrkii auttamaan parhaansa mukaan. ScrumMaster työskentelee asiakkaiden ja hallinnon kanssa tunnistaakseen ja määrittääkseen tuoteomistajan sekä muodostaakseen Scrum-tiimin. Sen jälkeen ScrumMaster työskentelee tuoteomistajan sekä Scrum-tiimin kanssa luodakseen tuotteen kehitysjonon (product backlog), mihin sprintit pohjautuvat. ScrumMaster

suunnittelee ja aloittaa sprintit tiiminsä kanssa. Sprintin aikana hän johtaa kaikki päivittäiset palaverit ja on vastuussa siitä, että ongelmia ratkaistaan ja päätöksiä tehdään. Hän on myös vastuussa kehitysjonon lyhentämisestä työn edetessä. (Schwaber & Beedle 2002, 31-32.)

Tuotteen kehitysjono on lista kaikista ominaisuuksista, toiminnallisuuksista, teknologioista, parannuksista ja bugien korjauksista jotka ovat osana tuotteeseen tehtäviä muutoksia tulevissa julkaisuissa. Kaikki mikä edustaa tuotteeseen tehtävää työtä on sisällytetty tuotteen kehitysjonoon. Aluksi tuotteen kehitysjono on keskeneräinen alustava lista asioita joita tuote tai järjestelmä tarvitsee. Tuotteen kehitysjonossa tarvitsee aluksi olla vain sen verran vaatimuksia, että ensimmäinen kolmenkymmenen päivän sprintti voidaan aloittaa. Kehitysjono on dynaaminen ja muuttuu lähes jatkuvasti, tarkoituksena pitää tuote hyödyllisenä ja kilpailukykyisenä. Tuotteen kehitysjono on niin kauan olemassa kuin itse tuotekin. Tuotteen kehitysjono on järjestetty tärkeyden mukaan. Mitä korkeammalla kehitysjonossa oleva asia on, sitä kiireellisempi ja arvokkaampi se on työn kannalta. Kehitysjono sisältää lisäksi ongelmia, jotka myös ovat tärkeysjärjestyksessä. Ongelma pitää ratkaista ennen kuin yhtä tai useampaa kehitysjonon kohtaa voidaan työstää. Ongelma saattaa vaatia muuttamista kehitysjonon kohdaksi tarvittavina toiminnallisuuksina tai teknologioina. Tuoteomistaja on vastuussa ongelmien muuttamisesta työksi, joka voidaan valita sprinttiin. Tuoteomistaja on yksittäinen henkilö joka on vastuussa tuotteen kehitysjonon hallinnoimisesta. Tuoteomistaja on myös projektin virallinen vastuuhenkilö. (Schwaber & Beedle 2002, 32-34.)

ScrumMaster ja Scrum-tiimi käyvät läpi tuotteen kehitysjonon. Tiimi omistautuu tekemään toimivan tuotteen valituista kehitysjonon kohdista. Scrum-tiimi tekee tämän jokaista sprinttiä varten. Tiimillä on auktoriteetti tehdä mitä tahansa saadakseen työn tehtyä. Tiimin koon tulisi olla seitsemän tai noin viidestä yhdeksään henkilöä. Kolmenkin henkilön tiimi voi hyötyä Scrumista, mutta tiimin pieni koko vähentää mahdollista interaktiota ja tuottavuutta. Kahdeksaa henkilöä suuremmat tiimit eivät toimi kovin hyvin. Tiimin tuottavuus laskee ja Scrumin ohjausmekanismit muuttuvat kömpelöiksi. Päivittäisten palaverien vetäminen voi tulla liian vaikeaksi ScrumMasterille ja suurikokoinen tiimi tekee prosessista turhan monimutkaisen. Suuri määrä työntekijöitä on suositeltava jakaa useaksi tiimiksi. Ensimmäinen tiimi valitsee

työstettävät kohdat kehitysjonosta ja aloittaa sprintin. Seuraava tiimi tekee vastaavasti jäljellä olevan kehitysjonon puitteissa. Päivittäisten palaverien jälkeen tiimien ScrumMasterit pitävät vielä lisäksi oman päivittäisen palaverinsa, koordinoiden yhdessä projektikokonaisuutta. (Schwaber & Beedle 2002, 35-37.)

5 VERSIONHALLINTA

Tässä luvussa kerron hieman versionhallinnasta ja sen muodoista. Kirjoitan myös lyhyesti versionhallintatuotteiden historiasta.

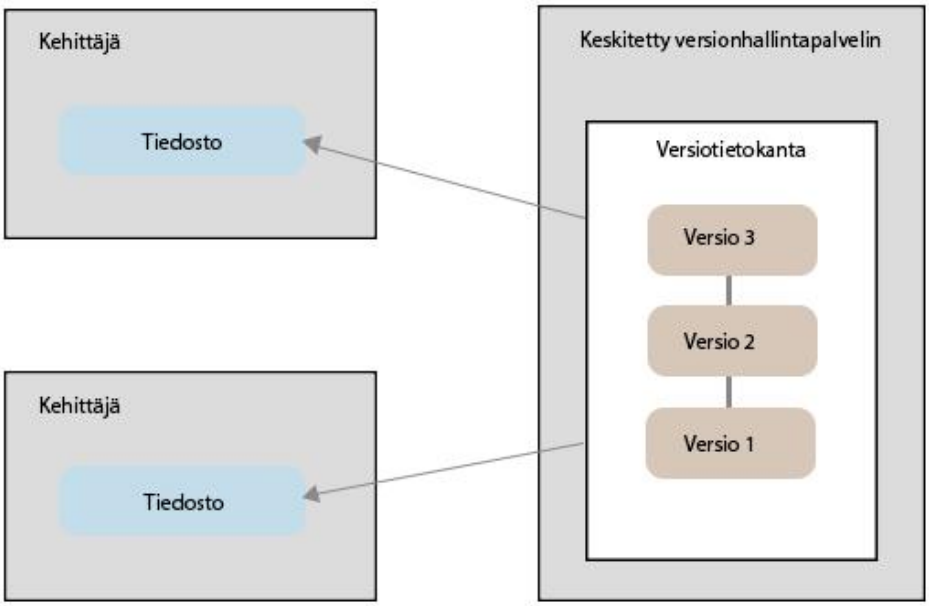
5.1 Mitä on versionhallinta?

Versionhallinta on järjestelmä, jonka tarkoitus on pitää kirjaa muutoksista tiedostoon tai tiedostojoukkoon, minkä ansiosta tiettyä versiota voidaan tarkastella tai hyödyntää myöhemmin. Versionhallintajärjestelmä mahdollistaa muutosten vertailun tietyltä ajalta, tiedostojen palauttamisen aiempaan versioon tai jopa koko projektin palautuksen aiempaan tilaan. Sillä myös näkee ja kuka on viimeksi muokannut jotain mikä saattaa aiheuttaa ongelmia, sekä milloin kyseinen muutos on tehty. Versionhallintajärjestelmä mahdollistaa tiedostojen helpon palautuksen jos niitä on vahingossa poistettu. (Chacon & Straub 2015, 27.)

Monet ihmiset käyttävän versionhallintamenetelmää missä he yksinkertaisesti kopioivat tiedostoja toiseen mahdollisesti aikaleimattuun kansioon. Lähestymistapa on yleinen koska se on helppo, mutta se on myös hyvin virhealtis. On helppoa unohtaa, missä kansiossa on ja ylikirjoittaa tai korvata tiedostoja mille ei näin pitänytkään tehdä. Ensimmäiset versionhallintajärjestelmät tehtiin tämän ongelman korjaamiseksi. (Chacon & Straub 2015, 27-28.)

Yksi kehitystyön ongelmista on yhteistyö eri järjestelmissä olevien kehittäjien kanssa. Keskitetty versionhallinta kehitettiin tästä syystä. Järjestelmällä on keskuspalvelin joka sisältää kaikki versioidut tiedostot ja asiakasohjelmat voivat kirjata tiedostoja ulos tästä keskusvarastosta (Kuva 3). Tämä oli versionhallintajärjestelmien standardi

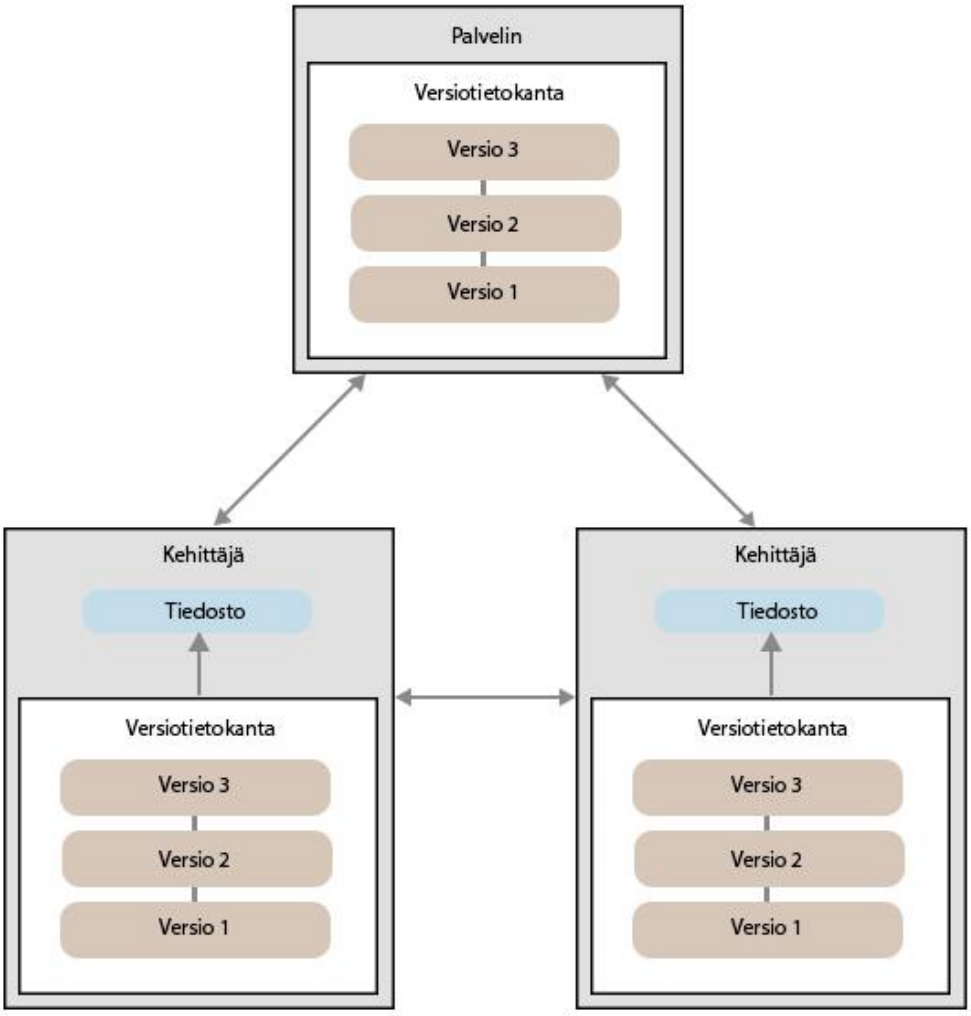
monen vuoden ajan. Mallilla on monia etuja, varsinkin verrattaessa paikallisiin versionhallintajärjestelmiin. Kaikki esimerkiksi tietävät jollain tasolla, mitä muut projektityöntekijät tekevät. Ylläpitäjät voivat hienosäädellä käyttäjäoikeuksia – on paljon helpompi ylläpitää keskitettyä versionhallintajärjestelmää kuin useita paikallisia versiotietokantoja. (Chacon & Straub 2015, 28-29.)



Kuva 3. Keskitetty versionhallintajärjestelmä (Chacon & Straub 2015, 29).

Keskitetyssä versionhallintajärjestelmässä on myös vakavia haittapuolia. Keskitetty palvelin on järjestelmän heikoin lenkki. Palvelimen ollessa alhaalla kukaan ei voi tallentaa tekemiään versioituja muutoksia. Jos palvelimen massamuisti korruptoituu eikä kunnollisia varmuuskopioita ole, projektin koko muutoshistoria on menetetty lukuun ottamatta pieniä yksittäisiä osia mitä työntekijöillä on itsellään. Paikallinen versiohallinta pitää sisällään saman riskin. Aina kun projektin koko muutoshistoria on vain yhdessä paikassa, on mahdollista menettää kaikki. (Chacon & Straub 2015, 29.)

Jaettu versionhallinta korjaa keskitetyn versionhallinnan pahimman haavoittuvuuden. Jaetussa versionhallinnassa asiakasohjelmat eivät kirjaa ulos vain tiedostoja niiden viimeisimmässä tilassa, vaan asiakasohjelmat peilaavat koko repositoriota (Kuva 4). Täten jos keskuspalvelin kaatuu, mikä tahansa asiakasohjelmarepositorioista voidaan palauttaa palvelimelle. Jokainen klooni on siis täysi varmuuskopio.



Kuva 4. Jaettu versionhallintajärjestelmä (Chacon & Straub 2015, 30).

5.2 Versionhallintajärjestelmien kehitys

Source Code Control System (SCCS) on luultavasti ensimmäinen versionhallintajärjestelmä saatavilla Unixille. Sen kehitti M. J. Rochkind 1970-luvun alkupuolella. SCCS:n tarjoamaa keskusvarastoa kutsuttiin repositorioksi ja tämä keskeinen konsepti on vieläkin käytössä. SCCS tarjosi myös yksinkertaisen lukituksen kehittämisen sarjallistamiseksi. Jos kehittäjä tarvitsi tiedostoja ajaakseen ja testatakseen ohjelmaa, voisi hän kirjata ne ulos lukitsemattomina. Tiedostoja muokatessa tuli ne kirjata ulos lukittuina – käytäntö jota Unixin tiedostojärjestelmä valvoi. Muutokset tehtyään kehittäjä kirjaisi tiedostot takaisin repositorioon, avaten samalla lukituksen. (Loeliger 2009, 4.)

Walter Tichy esitteli Revision Control Systemin (RCS) 1980-luvun alussa. RCS otti käyttöön myös deltakonseptin tilan säästämiseksi saman tiedoston eri versioissa. Jokaista versiota tiedostosta ei tallennettaisi erikseen vaan sen sijaan pidettäisiin vain tieto muutoksista, minkä perusteella tiedoston ensimmäisestä tai viimeisestä versiosta voitaisiin muutoslokia käyttäen saada mikä tahansa versio. Concurrent Version System (CVS) laajensi ja muokkasi RCS:n mallia hyvällä menestyksellä. CVS:n kehitti alun perin Dick Bruner vuonna 1986, mutta se luotiin uudelleen neljä vuotta myöhemmin. CVS tuli hyvin suosituksi ja oli monen vuoden ajan yleinen standardi avoimen lähdekoodin yhteisössä. CVS esitteli uuden paradigman lukitukselle. Aiemmat versionhallintajärjestelmät pakottivat kehittäjän lukitsemaan jokaisen muokattavan tiedoston, mikä taas pakotti muut kehittäjät odottamaan kunnes tiedosto muokkauksineen kirjattiin takaisin järjestelmään. CVS antoi jokaiselle kehittäjälle kirjoitusoikeudet kehittäjän omaan työstettävään kopioon. Tällä tavoin eri kehittäjien tekemät muutokset voitiin automaattisesti yhdistää, paitsi jos kehittäjät muokkasivat samoja rivejä. Tämä muutos tarkoitti sitä, että kehittäjät voisivat ohjelmoida samanaikaisesti. (Haikala & Märijärvi 2006, 263; Loeliger 2009, 5.)

CVS:n havaitut puutteet ja ongelmat johtivat uuteen versionhallintajärjestelmään. Vuonna 2001 julkaistu Subversion (SVN) tuli nopeasti kuuluisaksi vapaan lähdekoodin yhteisössä. Toisin kuin CVS, SVN tarjosi muutosten puskemisen jakamattomasti tarkoittaen että joko kaikki muutokset menevät läpi tai mikään muutos ei mene läpi. Subversion myös tuki versiohaaroja huomattavasti paremmin. (Loeliger 2009, 5.)

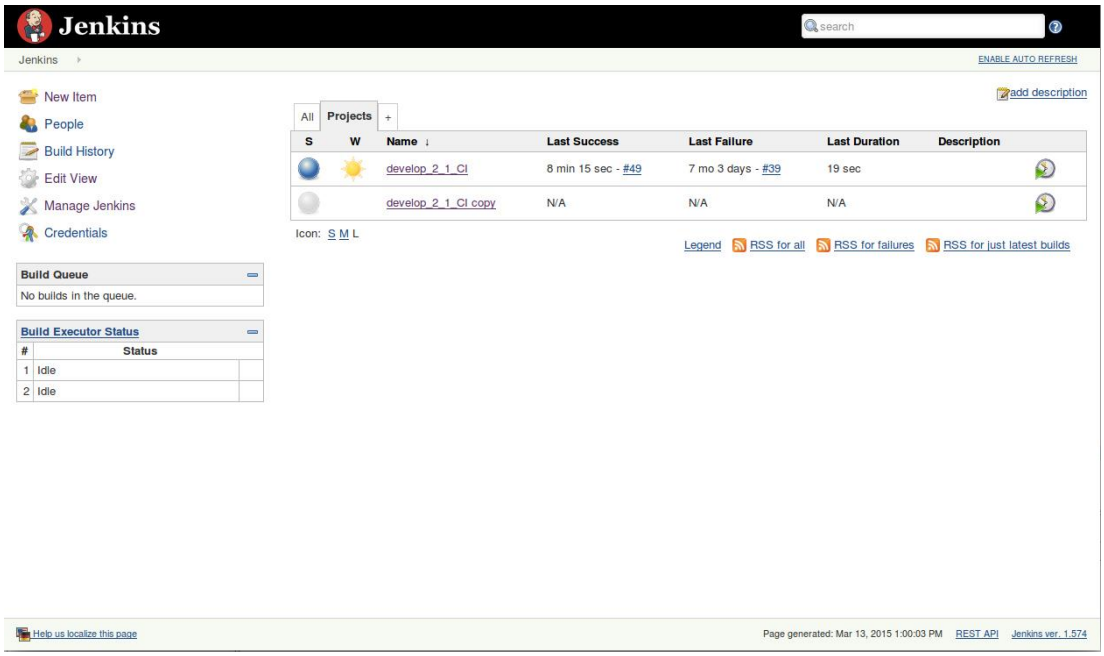
BitKeeper ja Mercurial olivat iso muutos edellä mainituista järjestelmistä. Molemmat luopuivat keskitetystä varastosta ja siirtyivät jaettuun malliin, missä jokainen kehittäjä sai oman jaettavan kopion. Mercurial ja Monotone lisäsivät sormenjälkenä toimivan tarkisteen, millä tiedoston sisältö voitaisiin tunnistaa uniikisti. Tiedostonimi on vain muodollisuus ja keino millä käyttäjä voi sen tunnistaa helpommin. Myös vuonna 2005 julkaistu Git hyödyntää näitä ominaisuuksia. (Loeliger 2009, 5.)

6 KÄYTETYT TYÖKALUT

Tässä luvussa käsittelen tärkeimpiä tämän opinnäytetyön käytännön osuudessa käytettyjä työkaluja. Näitä työkaluja ovat Jenkins, Ant ja Git. Käytin työn aikana paljon myös ohjelmointiympäristö Eclipseä, mutta koska Eclipse itsessään ei ollut työn kannalta välttämätön työkalu, en sitä tähän sisällytä. En myöskään kirjoita Oracle Express Edition -tietokannasta. Vaikka tuote onkin tärkeä osa järjestelmää integraatiotestien kannalta, ei se suoranaisesti vaikuttanut omaan työhöni muuten kuin tuotekohtaisen SQL-syntaksin käytön kannalta pienessä osassa työtä skeemoja luotaessa.

6.1 Jenkins

Jenkins, alun perin Hudson, on Javalla tehty avoimen lähdekoodin työkalu jatkuvaa integraatiota varten. Johtavassa markkina-asemassa olevaa Jenkinsiä käyttävät kaikenkokoiset tiimit erilaisiin projekteihin useilla eri kielillä ja teknologioilla mukaan lukien muun muassa .NET, Ruby, Groovy, Grails, PHP ja tietenkin Java. Jenkins on helppokäyttöinen ja sen käyttöliittymä on yksinkertainen sekä intuitiivinen lyhyen tutustumisen jälkeen (Kuva 5). Kokonaisuudessaan Jenkinsillä on hyvinkin loiva oppimiskäyrä. Jenkins ei kuitenkaan uhraa tehokkuutta tai laajennettavuutta käytettävyyteen, vaan työkalu on erittäin joustava ja helppo mukauttaa omiin käyttötarkoituksiin. Saatavilla on satoja avoimen lähdekoodin liitännäisiä moneen eri tarkoitukseen. Jenkinsin suosioon vaikuttaa paljon myös sen aktiivinen ja laaja yhteisö. Kehitystyö on nopeatempoista ja uusia julkaisuja tulee viikoittain uusilla ominaisuuksilla, virhekorjauksilla ja liitännäispäivityksillä. Jatkuvaa päivittämistä kammoava voi vaihtoehtoisesti käyttää noin kolmen kuukauden välein päivitettävää LTS-julkaisua. (Jenkinsin www-sivut 2014; Smart 2011, 3-4.)



Kuva 5. Jenkinsin käyttöliittymä päänäkymässä.

Jenkins syntyi ohjelmistokehittäjä Kohsuke Kawaguchin harrastusprojektista nimeltä Hudson, jonka hän aloitti vuonna 2004 työskennellessään yrityksessä Sun Microsystems. Sunin sisäiset tiimit alkoivat käyttämään Hudsonia omissa projekteissaan työkalun kehittyessä vuosien aikana. Alkuvuodesta 2008 Sun tunnusti työkalun laadun ja arvon, minkä seurauksena Kawaguchia pyydettiin kehittämään Hudsonia täyspäiväisesti tarjoten ammattimaista palvelua ja tukea. Vuoteen 2010 mennessä Hudsonista oli tullut johtava jatkuvan integraation ratkaisu yli 70 prosentin markkinaosuudella. Oracle osti yhtiön vuonna 2009. Vuoden 2010 lopussa Oraclen ja Hudsonin kehittäjäyhteisön välille syntyi ongelmia. Oracle halusi siirtyä tarkemmin hallittuun kehitysprosessiin hitaammalla julkaisuaikataululla, kun taas suurin osa Hudsonin kehitystiimistä halusi pysyä nopeassa, joustavassa ja yhteisökeskeisessä mallissa joka oli siihen asti toiminut hyvin. Tammikuussa 2011 Hudsonin kehitysyhteisö päätti vaihtaa projektin nimen Jenkinsiksi, haarauttaa se omaksi projektikseen ja jatkaa kehitystyötä sen parissa. Laaja enemmistö kehittäjistä ja liitännäisten tekijöistä seurasivat uuden projektin perässä. Haarautuksen jälkeen myös valtaosa käyttäjistä siirtyivät Hudsonista Jenkinsiin. (Smart 2011, 4.)

6.2 Apache Ant

Apache Ant on pieni Java-pohjainen ohjelma mikä on suunniteltu auttamaan ohjelmistokehitystiimejä kehittämään isoja ohjelmia automatisoimalla työläitä tehtäviä kuten koodin kääntäminen, testien ajaminen ja tulosten paketointi jakeluversioksi. Ant on suunniteltu järjestelmäriippumattomaksi, helppokäyttöiseksi, laajennettavaksi ja skaalattavaksi. Sitä voidaan käyttää pieneen henkilökohtaiseen projektiin tai laajaan, usean tiimin ohjelmistoprojektiin. Sen tarkoitus on automatisoida koko build-prosessi. (Loughran & Hatcher 2007, 5.)

James Duncan Davidsonin eli Antin alkuperäisen kehittäjän mukaan nimi Ant on akronyymi sanoista Another Neat Tool. Myöhemmin työkalun tarkoitusta on kuvailtu myös vertaamalla sitä muurahaisiin, kertoen niiden olevan hyviä rakentamaan ja pystyvän kantamaan moninkertaisesti oman painonsa. (Apache Antin www-sivut 2014.)

Ant luotiin alun perin vain olemaan työkalu Apache Tomcatin helpompaan kääntämiseen eri alustoilla. Tavoite saavutettiin ja muiden kehittäjien avulla työkalusta tuli tapa, jolla kaikki Apachen Java-projektit buildattiin. Ant levisi nopeasti myös muihin avoimen lähdekoodin projekteihin ja hiljalleen yleiseen käyttöön Java-kehittäjien keskuudessa. Vuonna 2000 ja vielä muutamaa vuotta myöhemmin Ant oli hieman epätavanomainen työkalu Java-kehittäjien keskuudessa. Nykyään on lähes odotettua löytää Antin build-tiedostoja Java-projektista lähdekoodin ja testien mukana. Ant käyttää XML-syntaksia, mikä on monelle kehittäjälle ennestään tuttu. Ant on myös laajennettavissa Javassa itsessään, mikä antaa mahdollisuuden käyttää alustan funktionalisuutta ja kolmannen osapuolen kirjastoja. Tämä myös tekee buildista nopean, sillä Java-ohjelmia voidaan ajaa samasta Java-virtuaalikoneesta kuin missä Ant on. (Apache Antin www-sivut 2014; Loughran & Hatcher 2007, 5-7.)

6.3 Git

Git on Linus Torvaldsin tekemä versionhallintajärjestelmä, minkä hän kehitti tukemaan Linux-ytimen kehitystä. Ohjelma kuitenkin osoittautui arvokkaaksi

työkaluksi laajassa valikoimassa projekteja. Työkalua käytetäänkin monissa projekteissa ja yrityksissä kuten Google, Microsoft ja Netflix. Git erottuu muista versionhallintaohjelmista eniten haarautusmallinsa ansiosta. Git sallii useita, toisistaan itsenäisiä paikallisia haaroja. Haarojen luominen, yhdistäminen ja poistaminen tapahtuu sekunneissa. (Gitin www-sivut 2014; Loeliger 2009, 1-2.)

Ennen Gitiä Linux-ydintä kehitettiin käyttäen kaupallista BitKeeper-versionhallintaohjelmaa, missä oli hienostuneita toimintoja mitä ei muissa silloisissa ilmaisissa ohjelmissa ollut. BitKeeperin omistava yhtiö päätti kuitenkin lisätä rajoitteita ohjelman ilmaisversioon keväällä 2005, minkä seurauksena Linux-yhteisö ymmärsi ettei BitKeeper ole enää oikea ratkaisu. Torvalds etsi vaihtoehtoja vältellen maksullisia ohjelmia, mutta löysi ilmaisista ohjelmista samat ongelmat ja rajoitteet joiden takia hän oli ne aiemminkin sivuuttanut. Ainoaksi vaihtoehdoksi jäi tehdä sopiva ohjelma itse ja monen kehittäjän yhteistyöllä syntyi Git. Git julkaistiin huhtikuussa 2005. (Loeliger 2009, 2-5.)

7 TOTEUTUS

Tässä luvussa kuvailen opinnäytetyön käytännön osuuden vaiheita. Tavoitteena oli saada soluohjaimen (CellController) kehitystiimille toimiva integraatiopalvelin. Järjestelmän pitää osata automaattisesti havaita muutokset versionhallintajärjestelmässä projektikohtaisen haaran osalta, kääntää ohjelmakoodi ja ajaa integraatiotestit tietokantaa vasten, sekä virhetilanteissa lähettää välittömästi ilmoitus muutoksen tehneelle kehittäjälle. Järjestelmän tulee tukea useita haaroja ja pystyä tekemään kaikki edellä mainittu useammalle projektille samanaikaisesti. Lisätavoitteina oli myös laadunvalvontaan ja seurantaraportteihin liittyvien liitännäisten asennus ja käyttö sekä katselmointityökalun liittäminen järjestelmään.

7.1 Ympäristö ja pohjustus

CIServerin alustana toimii virtuaalipalvelimelle asennettu CentOS. Palvelimella on asennettuna Oracle Express Edition 11g R2 -tietokanta, Javan versio 1.6 sekä

versionhallintaohjelma Git. Ennen järjestelmän valmistumista ja asentamista CIServerille oli Jenkinsistä useampi testiasennus, joilla rakensin ja muokkasin järjestelmää saadakseni sen toimimaan halutulla tavalla. Lisäksi asensin vastaavanlaisen ympäristön kotona virtuaalikoneelle, missä pystyin vapaammin kokeilemaan mahdollisia ratkaisuja ongelmiin. Kotiasennuksessa käytin yksinkertaista omaa koodia minkä kääntämiseen kului alle 15 sekuntia testeineen, mikä huomattavasti helpotti eri menetelmien kokeilua. Yrityksellä ollessani keskityin enemmän ratkaisujen soveltamiseen yrityksen koodiin ja saamaan projektin kääntymään Jenkinsissä oikein.

7.2 Jenkinsin asennus

Jenkins oli helppo asentaa sen verkkosivuilta löytyvien ohjeiden mukaan. Ennen palveluprosessin käynnistämistä olisi voitu tarpeen vaatiessa vaihtaa Jenkinsin oletusarvoinen porttinumero 8080 johonkin muuhun, tai konfiguroida Jenkins käyttämään salattua liikennettä. Jenkinsin asennus on toistaiseksi rajoitettu sisäverkkoon, joten toimenpiteitä ei tässä kohtaa katsottu aiheellisiksi. Edellä mainitut asetukset tulevat tiedostosta /etc/init.d/jenkins, kun taas Jenkinsin varsinaiseen toimintaan liittyvät asetukset löytyvät Jenkinsin sisältä.

7.3 Jenkinsin konfigurointi

Asennuksen jälkeen Jenkins käynnistettiin palveluprosessina, minkä jälkeen palvelu näkyi verkkosivuna sisäverkossa. Ennen konfigurointia täytyi vielä ladata ja asentaa erinäisiä liitännäisiä. Toimivuuden kannalta tärkeimmät liitännäiset ovat Ant Plugin, Git Plugin ja testng-plugin. Lisäksi oli asennettu myös näkymään kuvauskolumnin lisäävä liitännäinen, mikä helpottaa vapaan skeeman valintaa uudelle projektille. Asetuksissa määritettiin Javan sijainti sekä asetettiin oletusnäkymän tilalle kuvauskolumin sisältävä näkymä. Konfigurointiin sisältyi myös sähköpostiasetuksien määrittäminen, Apache Antin asentaminen Jenkinsillä sekä palvelun perustietojen muokkaus.

Versionhallintaohjelma Git käyttää SSH-yhteyttä, mitä varten täytyi Jenkinsille tehdä salasanaton SSH-avain. Avain lisättiin Git-palvelimelle luotettuihin yhteyksiin, minkä jälkeen Jenkins pystyy liitännäisen avulla odottamaan kääntämisen oikeuttavaa koodimuutosta. Muutoksen havaittuaan uusin versio koodista ladataan automaattisesti työkansioon ja suoritetaan määrätyt toiminnot.

7.4 Sähköposti-ilmoitukset

Aiemmin integraatiotestejä ajettaessa piti kehittäjän odottaa, kunnes työkone oli saanut testit ajettua. Työajan menetys saattoi olla haitallista varsinkin kiireisellä aikataululla. Jenkinsiä käytettäessä kehittäjä lähettää koodimuutokset versionhallintaan ja järjestelmä ajaa testit muualla, milloin kehittäjä voi jatkaa työskentelyä tai siirtyä seuraavaan tehtävään. Testit ajettuaan Jenkins lähettää tarvittaessa sähköposti-ilmoituksen joko pelkästään muutoksen tehneelle kehittäjälle tai koko tiimille.

Ongelmien syntyessä tieto saadaan nopeasti ja vika voidaan korjata. Sähköposti-ilmoitus ja sen vastaanottajat voidaan määrittää projektikohtaisesti tai pitää samana projektista toiseen. Konfiguroitavissa on myös se, milloin ilmoituksia tulisi lähettää. Virheen syntyessä tai sen korjautuessa on hyvä saada tieto nopeasti, jotta kehitystyö voi jatkua. Vastaavasti monen perättäisen käännöksen mennessä läpi virheettä ei ilmoitus ole yleensä tarpeellinen. Muutoksen tehneen kehittäjän sähköpostiosoite saadaan versionhallinnalta, kunhan kehittäjän Git on konfiguroitu oikein.

7.5 Apache Ant -skriptit

Yksi isoimmista osista työtä oli Ant-skriptien tekeminen ja muokkaus. Skriptejä oli valmiiksi koodin kääntämiseen sekä tuotteen asentamiseen ja poistamiseen. Aluksi haasteena oli sovittaa skriptit toimimaan halutulla tavalla Jenkinsin kanssa. Lähes kaikki tähän liittyvät muutokset olivat polkurakenteiden muokkaamista. Lisähaastetta syntyi siitä, etten ollut käyttänyt Antia aikaisemmin vaan tutustuin siihen tämän työn aikana. Skriptit itsessään ovat rakenteeltaan mielestäni melko yksinkertaisia, mutta tarvittavien elementtien käyttö oikeassa järjestyksessä oli alkuun haasteellista.

Tähänkin auttoi kotona ollut asennus, minne pystyin itse rakentamaan yksinkertaisia skriptejä sekä ymmärtämään niiden toiminnan. Moni ratkaisu ongelmiin löytyikin tällä tavalla, minkä jälkeen piti vain soveltaa ratkaisu monimutkaisempaan ympäristöön.

7.5.1 Integraatiotestien kääntäminen ja ajo

Valmiit Ant-skriptit hoitavat tuotteen kääntämisen ja asentamisen tuotantoympäristöön. Testejä oli aiemmin ajettu käsin kehittäjien koneilta, mistä johtuen testit kääntävä ja ajava skripti piti tehdä itse. Soluohjain voidaan tästä näkökulmasta purkaa viiteen osaan, mihin kuuluu pohja eli CellBase ja neljä moduulia erilaisiin robotteihin. Tarkoituksena oli saada ajettua minkä tahansa tai jokaisen moduulin testit automaattisesti. Jokaisella moduulilla on omanlaisensa testitietokanta, mitä vastaan testejä ajetaan. Tietokanta alustetaan aina moduulista löytyvän alustusmetodin kautta ohjelmallisesti. Päädyin ratkaisuun, jossa testien kääntäminen ja ajo suoritetaan kolmessa vaiheessa: moduulin testiosion kääntäminen, tietokannan alustus ja testien ajo.

Testien kääntämisen suorittava skripti oli suhteellisen yksinkertainen tehdä. Nimen lisäksi ainoa ero moduulikohtaisissa skriptilohkoissa olikin yksi polkumuuttuja (Kuva 6). Skripti kutsuu kääntäjää annetuilla parametreillä ja lopputuloksena on käännetty moduulin testauskoodi.

```
<target name="buildTyrePickIntegrationTests" depends="dist">
    <echo message="-- Building tests --"/>
    <delete includeEmptyDirs="true">
        <fileset dir="${build.dir}">
            <include name="*/**"/>
        </fileset>
    </delete>
    <javac srcdir="${test.tyre.integration.src.dir}:${test.base.integration.src.dir}:${test.base.src.dir}"
           destdir="${build.dir}"
           classpathref="test.classpath"
           includeantruntime="true"
           debug="true"
           debuglevel="lines,source"
           encoding="iso-8859-1"/>
</target>
```

Kuva 6. Moduulin integraatiotestien kääntäminen.

Testien kääntämisen jälkeen seuraava vaihe on alustaa tietokanta. Tämä tapahtuu kutsumalla moduulin metodia BuildTestCell, joka alustaa tietokannan (Kuva 7).

Metodi alustaa tietokannan ja täyttää sen moduulin integraatiotestausta varten luodulla testidatalla, simuloiden tuotannossa olevaa järjestelmää.

```
<target name="init_tyrepick_testdb" depends="dbprep.create_user, intr_set_db_schema, dist">
    <java classname="fi.cimcorp.cell.integrationtest.BuildTestCell" classpathref="test.classpath">
        <classpath>
            <pathelement path="${testdb.tyre.init.class.dir}" />
            <pathelement path="${testdb.base.dir}" />
        </classpath>
    </java>
</target>
```

Kuva 7. Moduulin testitietokannan alustaminen.

Viimeinen vaihe oli ajaa moduulikohtaiset testit. Jokaisella moduulilla on kaksi testikokoelmaa, joista kummatkin pitää ajaa (Kuva 8). Testikokoelmien muistivaatimukset kuitenkin vaihtelivat, mistä aiheutui aluksi ongelmia. En tiennyt, onko kyseessä virhe skriptissä vai jokin vakavampi ongelma. Päädyin lopulta määrittämään jokaiselle testikokoelman ajolle tarvittavan määrän muistia, mikä ratkaisi ongelmat. Testitulokset muodostetaan aina kullekin TestNG-kutsulle, mistä johtuen jokaiselle moduulille syntyy kaksi tulosraporttia. Liitännäinen kuitenkin osaa yhdistää useamman raportin tulokset yhdeksi kokonaisuudeksi, joten tästä ei aiheutunut ongelmia.

```
<target name="test_tyrepick_intr" depends="init_tyrepick_testdb">
    <echo message="-- Running tests --"/>
    <testng outputDir="${testng.report.dir}/tyrepick/" haltOnFailure="false" workingDir="${testng.tyre.testsuites.dir}">
        <jvmarg line="-Xmx256m" />
        <classpath refid="test.classpath" />
        <xmlfileset dir="${testng.tyre.testsuites.dir}" includes="testng-integration.xml"/>
    </testng>
    <testng outputDir="${testng.report.dir}/tyrepick/2/" haltOnFailure="false" workingDir="${testng.tyre.testsuites.dir}">
        <jvmarg line="-Xmx512m" />
        <classpath refid="test.classpath" />
        <xmlfileset dir="${testng.tyre.testsuites.dir}" includes="testng-integration-orders.xml"/>
    </testng>
</target>
```

Kuva 8. Moduulin testikokoelmien ajaminen.

7.5.2 Parametrisointi

Soluohjain oli tässä vaiheessa käännettävissä ja integraatiotestit ajettavissa. Käytettävä skeema olisi kuitenkin pitänyt vaihtaa käsin, jos käännettäviä projekteja olisi halunnut enemmän kuin yhden. Skeeman käsin muuttaminen koodiin tai Ant-skriptiinkään ei kuitenkaan ollut millään tavalla hyvä vaihtoehto, vaan piti tämä saada automatisoitua mahdollisimman pitkälle. Ratkaisuksi löytyi parametrisoitu projekti. Jenkins lähettäisi projektikohtaiset parametrit eteenpäin, jolloin Ant-skriptit voisivat niiden perusteella

tehdä tai olla tekemättä jotain. Testauksen kannalta tärkein parametri on käytettävä skeema. Varsinainen koodi lukee käytettävän skeeman ominaisuustiedostosta, minne taas Ant-skripti ylikirjoittaa oletusarvoisen skeeman parametreillä välitettyyn (Kuva 9). Skripti myös varmistaa, että parametrinä välitetty skeema ei ole tyhjä merkkijono.

```
<target name="intr_set_db_schema">
    <fail message="Schema not set">
        <condition>
            <equals arg1="${DB.SCHEMA}" arg2=""/>
        </condition>
    </fail>

    <replaceregexp file="${config.dir}/integrationtest.properties"  match="db.username(.*)"
                                                replace="db.username=${DB.SCHEMA}" byline="true"/>
</target>
```

Kuva 9. Parametrillä välitetyn skeeman kirjoitus ominaisuustiedostoon.

Parametrisointia täytyi vielä laajentaa, kun huomasin ettei asennus enää onnistunutkaan virtualisoituun tuotantoympäristöön polkumuutoksista johtuen. Suurin osa Ant-skripteihin tehdyistä polkumuutoksista vaihdettiin takaisin alkuperäisiin arvoihin, mutta tällä kertaa muuttujien kautta. Muuttujien arvot voidaan suoraan korvata parametreinä välitetyillä arvoilla, milloin alkuperäiset kovakoodatut polut ovat oletusarvoisia. Tuote pystyttiin näin ollen taas asentamaan oikein sekä kääntyi vielä Jenkinsissäkin, kunhan suhteelliset polut välitettiin parametreinä.

7.5.3 Skeeman luominen tietokantaan

Moduulikohtaisia integraatiotestejä varten alustetaan aina tietokanta. Useita projekteja täytyy pystyä testaamaan samanaikaisesti, joten jokaiselle projektille määritetään oma skeema. Skeemojen luominen projektikohtaisesti on automatisoitu Ant-skriptillä (Kuva 10), mikä vastaanottaa käytettävän skeeman parametrinä Jenkinsiltä. Koska testejä ajetaan Oracle-tietokantaa vasten, täytyi tuotekohtaiset ominaisuudet ottaa huomioon. Oracle-tietokannassa käyttäjällä voi olla vain yksi skeema, millä on sama nimi kuin käyttäjälläkin (Oraclen www-sivut 2015). Käytännössä siis jokaiselle projektille on luotava oma käyttäjänsä, jotta projektien testejä voitaisiin ajaa eri skeemoissa. Käyttäjiä olisi voinut luoda useammallakin tavalla, mutta päädyin osaamiseni ja aikataulun rajoissa luomaan käyttäjät Ant-skriptillä.

```
<target name="drop_user">
    <sql driver="${sql.driver}"
         url="${sql.url}"
         userid="${sql.user]"
         password="${sql.pass}"
         delimiter="/"
         delimitertype="row"
         classpathref="ojdbc.dir">
        BEGIN
            EXECUTE IMMEDIATE ('DROP USER ${test.user} CASCADE');

            EXCEPTION
                WHEN OTHERS THEN
                    DBMS_OUTPUT.put_line('User does not exist');

        END;
        /
    </sql>
</target>

<target name="create_user" depends="drop_user">
    <sql driver="${sql.driver}"
         url="${sql.url}"
         userid="${sql.user}"
         password="${sql.pass}"
         classpathref="ojdbc.dir">
        CREATE USER ${test.user} IDENTIFIED BY ${test.pass};
        GRANT CONNECT, RESOURCE, CREATE TABLE, CREATE SEQUENCE TO ${test.user};
    </sql>
</target>
```

Kuva 10. Skeeman luonti Ant-skriptillä.

Aluksi tein Microsoft SQL Server -osaamiseni pohjalta proseduurin tehtävää varten, jota sitten kutsuttaisiin Ant-skriptillä. Proseduuri tarkisti onko käyttäjä valmiiksi olemassa ja tarvittaessa poistaisi sen ennen uuden luomista. Tein ja testasin proseduuria Microsoftin tuotteella, joka oli asennettuna omalla tietokoneellani. En kuitenkaan saanut proseduuria toimimaan Oraclen syntaksilla useankaan tunnin yrittämisen jälkeen. Oraclen dokumentaation ja esimerkkien perusteella tekemäni vähemmän elegantti ratkaisu kuitenkin toimi sekä omissa testeissäni että Cimcorpilla, joten otin sen käyttöön. Lopputuloksena on edelleen projekikohtaisesti määritetty skeema, mihin lisätään integraatiotesteissä käytettävä testidata tauluineen.

7.6 Projektin konfigurointi

Työ lähestyi loppuaan, mutta ongelmiakin vielä löytyi. Jostain syystä integraatiotestejä ei voitu ajaa ensimmäisen kääntämisen jälkeen. Havaitsin, etteivät käännetyt testit jostain syystä siirtyneet ajoissa oikeaan kansioon jotta integraatiotestit olisi voitu ajaa. Ratkaisuksi valitsin useamman korjausyrityksen jälkeen kahden Ant-kutsun käyttämisen jokaiselle moduulille. Ensimmäinen kutsu kääntää testit ja alustaa tietokannan, minkä jälkeen toinen kutsu ajaa integraatiotestit. Ratkaisu ei ollut mielestäni paras mahdollinen ja tuplasi tarvittavat Ant-kutsut projektissa. Aikaa ei kuitenkaan ollut enää tarpeeksi siihen, että olisin voinut tutkia ja mahdollisesti korjata ongelman paremmin. Projektin kääntämiseen kuluvaan aikaan ratkaisu ei kuitenkaan merkittävästi vaikuttanut.

Projektin konfigurointi oli nyt kuitenkin saatu vaiheeseen, missä ohjeistuksen pystyi kirjoittamaan tarkasti. Käytetään neljää parametriä, skeemalle yksi ja poluille kolme. Polut pysyvät samoina projektista toiseen kuten moni muukin asetus, joten projekteja on helppo ja nopea lisätä kopioimalla ennestään olemassa oleva projekti, minkä jälkeen konfiguroinnista muutetaan vain tarpeelliset kohdat kuten nimi, skeema ja haara. Muitakin asetuksia voidaan muuttaa projektikohtaisesti tarvittaessa. Kahdeksan Ant-kutsua pysyvät samoina ellei moduulien testausta jätetä pois. Integraatiotestien tulosten raportointi on vastaavasti usein samoilla asetuksilla. Versionhallinnasta tarkistetaan mahdolliset koodimuutokset tunnin välein sekä myös keskiyöllä.

8 YHTEENVETO

Valmis järjestelmä täyttää olennaiset vaatimukset. Projekteja voi olla useampi rinnakkain Jenkinsin ja laitteiston rajoissa. Eri projektien integraatiotestejä voidaan ajaa samaan aikaan, mikä olikin yksi tärkeimmistä päämääristä työssä. Yhden projektin kääntäminen ja neljän moduulin integraatiotestien ajaminen vie aikaa noin neljästäkymmenestä viiteenkymmeneen minuuttiin. Aika ei kasvanut ainakaan kahden projektin rinnakkaisesta kääntämisestä. Työn aikana ohjelmakoodiin täytyi tehdä

pieniä muutoksia, mutta tärkeimmät muutokset ja valtaosa työstä keskittyi Ant-skripteihin.

Kaikkia toivottuja ominaisuuksia en ehtinyt työn aikana toteuttamaan, mutta puutteet jäivät onneksi vain toivottuihin lisätoiminnallisuuksiin, mistä olisin silti halunnut ehtiä ainakin osan toteuttamaan. Vaaditut ja tärkeimmät ominaisuudet kuten integraatiotestauksen automatisointi ja useamman projektin testien ajaminen kuitenkin toimivat halutulla tavalla, joten olen suhteellisen tyytyväinen lopputulokseen. Työtä aloittaessani oli lähes kaikki uutta, joten opin paljon työn aikana. Työ oli haasteellinen mutta myös palkitseva, erityisesti silloin kun pitkään vaivanneet ongelmat sai ratkaistua.

Työkaluja oli mielestäni mukava käyttää ja niiden hyödyllisyyden huomasi jo työn alussa. Jenkinsin käyttäminen oli mielestäni helppoa ja melko suoraviivaista, vaikka muutaman kerran pitikin turvautua verkosta löytyviin ohjeisiin. Ant ja Git vaativat hieman enemmän opettelua ennen kuin niiden käyttäminen alkoi sujumaan luontevasti, mutta eivät kuitenkaan missään vaiheessa aiheuttaneet minulle hämmennystä.

Järjestelmään jäi myös parannettavaa. Koodikattavuustyökaluja käyttäen sekä muiden liitännäisten avulla järjestelmästä saisi yritykselle vielä paremman työkalun jatkuvaan integraatioon. Opin itse paljon sekä aiheesta että työssä käytetyistä työkaluista ja aionkin käyttää niitä jatkossa omissa harrastusprojekteissani.

LÄHTEET

Apache Antin www-sivut. 2014. Viitattu 15.11.2014.
http://ant.apache.org/faq.html

Chacon, S. & Straub, B. 2015. Pro Git, Second Edition. Apress. Viitattu 10.01.2015
https://progit2.s3.amazonaws.com/en/2015-02-04-10986/progit-en.319.pdf

Fowler, M. 2006. Continuous Integration. Viitattu 3.10.2014.
http://www.martinfowler.com/articles/continousIntegration.html

Gitin www-sivut. 2014. Viitattu 18.10.2014.
http://git-scm.com/about

Holcombe, M. 2008. Running an Agile Software Development Project. Hoboken: John Wiley & Sons, Inc.

Jenkinsin www-sivut. 2014. Viitattu 22.10.2014.
https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins

Jorgensen, P. 2008. Software Testing: A Craftsman's Approach, Third Edition. Boca Raton: Auerbach Publications.

Kawalerowicz, M. & Berntson, C. 2011. Continuous Integration in .NET. Stamford: Manning Publications Co.

Loeliger, J. 2009. Version Control with Git. Sebastopol: O'Reilly Media Inc.

Loughran, S. & Hatcher, E. 2007. Ant in Action. Greenwich: Manning Publications Co.

Haikala, I. & Märijärvi, J. 2006. Ohjelmistotuotanto. 11. p. Jyväskylä: Talentum Media Oy.

Oraclen www-sivut. 2015. Viitattu 15.02.2015.
http://docs.oracle.com/cd/B19306_01/server.102/b14220/schema.htm

Patton, R. 2006. Software Testing, Second Edition. Indianapolis: Sams Publishing.

Schwaber, K. & Beedle, M. 2002. Agile Software Development with Scrum. Upper Saddle River: Prentice Hall, Inc.

Smart, J. 2011. Jenkins: The Definitive Guide. Sebastopol: O'Reilly Media Inc.